Supporting Multi-View Development for Mobile Applications

Scott Barnett[a], Iman Avazpour[a], Rajesh Vasa[a], John Grundy[b]

[a]*Deakin Software and Technology Innovation Lab (DSTIL), School of Information Technology, Faculty of Science, Engineering and Built Environment, Deakin University, Burwood, Victoria 3125, Australia*
[b]*Faculty of Information Technology, Monash University, Monash, Victoria 3800, Australia*

Abstract

Interest in mobile application development has significantly increased. The need for rapid, iterative development coupled with the diversity of platforms, technologies and frameworks impacts on the productivity of developers. In this paper we propose a new approach and tool support, Rapid APPlication Tool (RAPPT), that enables rapid development of mobile applications. It employs Domain Specific Visual Languages and Modeling techniques to help developers define the characteristics of their applications using high level visual notations. Our approach also provides multiple views of the application to help developers have a better understanding of the different aspects of their application. Our user evaluation of RAPPT demonstrates positive feedback ranging from expert to novice developers.

Keywords: Visual Notation, Mobile App Development, Domain Specific Languages, Code Generation

1. Introduction

Mobile application (app) development has exploded. In 2014 the Google Play Store doubled its number of apps[1] and in December 2014 over 40000

Email addresses: `scott.barnett@deakin.edu.au` (Scott Barnett), `iman.avazpour@deakin.edu.au` (Iman Avazpour), `rajesh.vasa@deakin.edu.au` (Rajesh Vasa), `john.grundy@monash.edu` (John Grundy)

[1]http://blog.appfigures.com/app-stores-growth-accelerates-in-2014/

apps were submitted to the iTunes App Store[2]. Despite the popularity, app development is a tedious process as it requires copious amounts of code to be written using tools that lack support for high level abstractions. Modeling techniques such as Domain Specific Visual Languages (DSVL) simplify development of mobile apps by abstracting away the details, hence improving developer productivity. On the other hand, downsides from using modeling techniques consist of rigidity in the generated output and the lack of flexibility for specifying custom functionality. Mobile app development follows a user-centered design process where real-world human experience is central to the design process [1]. Hence, developers must focus on building a great user experience and cannot afford to be limited by the restrictions of a model based tool.

Typically an IDE consists of a code view and a designer for specifying the UI components of a screen. From these views developers cannot easily grasp the navigation flow of the app, identify where the resources provided by a single API are used, or clearly analyze the event handling for a single screen. The information for each of these concerns is embedded in the source code requiring developers to browse through the code. This lack of transparency restricts communication as not all stakeholders understand code.

Early high fidelity prototyping helps with communication of ideas. Due to the user centered aspect of mobile app development many developers choose to prototype. These prototypes often focus on the visual aspects of the mobile app striving to resemble the final app as closely as possible. Once the visual aspects of an app have been agreed upon these prototypes are discarded requiring developers to replicate the prototype in code. Prototyping other non-visual aspects of an app is rarely done due to the cost.

To aide professional app developers, this paper presents our approach in Rapid APPlication Tool (RAPPT) [2], a framework for multi-view development of mobile apps. RAPPT leverages model driven techniques to bootstrap mobile app development through use of a Domain Specific Textual Language (DSTL) and a Domain Specific Visual Language. It provides multiple views to developers ranging from detailed view (code) to abstract view (e.g. page navigation). These multiple views help developers implement mobile app concepts at different levels of abstraction improving the customizability of

[2]http://www.statista.com/statistics/258160/number-of-new-apps-submitted-to-the-itunes-store-per-month/

the app. These multiple views also improve communication between stakeholders by presenting a snapshot of the application for a specific concern. Key contributions of this paper are:

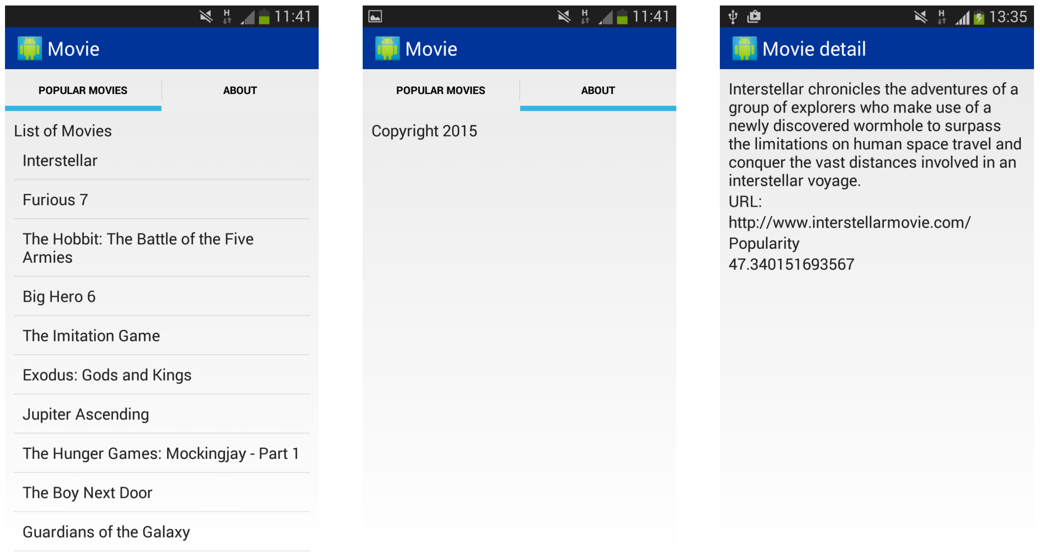
- A classification of the different levels of abstraction present for mobile app development.
- A multi view system is introduced that enables developers to work on different abstraction levels synchronously.
- We provide tool support for the fast development of mobile app prototypes using a multi-view modeling approach.

This paper is organized as follows: Section 2 provides a motivating scenario for this work. Next, we present the related work in Section 3. In Section 4 we provide background information on the different abstractions used in the development of mobile apps. Section 5 describes our approach including the various elements of the framework and concepts in the DSVL. Following that is an overview on our tool implementation RAPPT in Section 6. Section 7 provides details of our user evaluation and experiment setup and finally Section 8 concludes the paper and provides avenues of future work.

2. Motivation

Our primary motivation for this research comes from experience with developing mobile apps. In particular, dealing with the inadequate tools provided to professional developers when starting a new app. As a way to illustrate the mobile app development process and the tools used we present a motivating example for the development of a data-intensive mobile app, MovieDBApp. A data-intensive mobile app is an app that is predominantly focused on fetching, formatting and rendering data retrieved from a webservice.

Consider Peter, a mobile application developer, who is tasked with the development of MovieDBApp. MovieDBApp shows a list of popular movies that when selected enable a user to navigate to a screen showing the details of that movie. This app consists of three screens a *Popular Movies* screen for displaying the list of popular movies, an *About* screen to display copyright information, and a *Movie Detail* screen to display the details of a selected movie. Screenshots for these screens are shown in Figure 1. The content to



List of Movies

- Interstellar
- Furious 7
- The Hobbit: The Battle of the Five Armies
- Big Hero 6
- The Imitation Game
- Exodus: Gods and Kings
- Jupiter Ascending
- The Hunger Games: Mockingjay - Part 1
- The Boy Next Door
- Guardians of the Galaxy

Copyright 2015

Interstellar chronicles the adventures of a group of explorers who make use of a newly discovered wormhole to surpass the limitations on human space travel and conquer the vast distances involved in an interstellar voyage.
URL:
http://www.interstellarmovie.com/
Popularity
47.340151693567

Figure 1: The *Popular Movies*, *About*, and *Details* screens for MovieDB, an app based on the MovieDB API.

be displayed by this app will be provided by the freely accessible MovieDB API[3]. Peter's app includes the following requirements:

1. A tabbar for navigation between the *Popular Movies* and *About* screens. Tabbar is a Mobile UI Pattern that displays tabs near the top of the screen for navigation (see Figure 1). The user can also swipe the screen to navigate between tabs.
2. Authentication for connecting to the Movie DB API e.g specify an authentication key.
3. Display a list of the popular movies from the Movie DB API on the *Popular Movies* screen.
4. Navigate to the *Details* screen from the *Popular Movies* screen by selecting a list item.
5. Pass the identifier for a movie when clicking on a list item in the *Popular Movies* screen and pass it to the *Details* screen so the details for the correct movie can be fetched.

[3]http://docs.themoviedb.apiary.io/

6. Fetch the details for a selected movie from the MovieDB API and render the results to the screen on the *Details* screen.
7. Display a copyright message on the *About* screen.

To build this app, Peter needs to write the code for the tabbar specifying the tabs for each screen, configure the navigation for each tab, handle navigation to the tab screen and create the animation for swiping between tabs. In order to connect to the MovieDB API and handle authentication, he must write code for the API requests, model the data returned by API calls, handle errors, check network connectivity, authenticate with the MovieDB API and ensure best practices for concurrency on a mobile platform. For the *Popular Movies* screen, Peter needs to create the following: layout files for the UI, connection between data returned from API call and the UI components, event handlers for selecting a list item to navigate to the *Details* screen, and implement navigation pattern tabbar. The *Details* screen needs to accept the parameters passed from the *Popular Movies* screen, pass that parameter to an API call to fetch that data for the selected movie and render the movie details to the screen. The *About* screen needs to display a static message to the user. In addition to these tasks Peter needs to configure the build system, adding any dependencies, add logging code, create styles, debug, follow software engineering best practices such as ensuring maintainability and performance, while meeting stringent deadlines.

3. Related Work

Many of the current tools are aimed at inexperienced developers or non-technical experts [3, 4, 5, 6, 7, 8]. These approaches hide many of the implementation details described above from the end users [9] enabling them to focus on higher level constructs. While this makes programming more palatable, end users cannot customize how these apps address the intricate concerns of developing mobile apps [10, 11, 12, 13]. For example, Peter has specific requirements about where to fetch data from (i.e. the MovieDB API) and how to authenticate the app with the API using an API key. As such these approaches are not suited for use by a professional mobile app developer.

Multiple Model Driven Development (MDD) approaches for building mobile apps have been developed [9, 14, 15, 16, 17, 18]. These approaches commonly focus on MDD as a cross-platform approach to mobile app development. A common emphasis of these cross-platform approaches is on modeling a common subset of features that can then generate code for multiple target platforms. The modeling approaches often present a single way of modeling a mobile app and do not provide multiple views of a mobile app as in our proposed solution. For example, describe every aspect of an app by using a DSL [14]. In addition these MDD approaches tradeoff flexibility for productivity by abstracting away many of the implementation details. Data-intensive mobile apps share a lot of functionality yet have very unique UI and interactions. To support building unique apps a tool should support a high degree of customization. Recent work has also focused on using MDD for building context aware applications [19, 20]. These approaches focus on a subset of the functionality available for mobile apps where as RAPPT is designed around building data intensive applications.

Typically the app development process begins by designing the UI and User Experience with the client before these ideas are refined into mock-ups by a designer. Tools for prototyping [21, 22, 23] can greatly help with the evaluation of ideas especially concerning the navigation flow through an app. Once this process has been done developers have to start from scratch to implement the agreed upon navigation flow – current tools produce throwaway prototypes. In addition, implementing a prototype that can be reused by developers is a time intensive and costly process. What is needed is a tool that can be used for rapid prototyping but generates apps that can be built upon by developers when realizing the final app.

User Experience plays a crucial role in the development of mobile apps due to the influence of user reviews and rankings of the app store. Designers and developers need to pay careful attention to the usability of their apps. Modeling different components of an app such as the navigation flow is well suited to and frequently represented as a graphical visualization. Developers often spend a lot of time programming using text and is so more familiar to them for describing event handlers, data flows and UI bindings. Researchers have found multi-view approaches to be beneficial for addressing multiple concerns when building multiagent systems [24], in embedded systems engineering [25], and in the design of cyber-physical systems [26]. This shows that a multi-view approach is suitable for addressing the specific concerns of mobile app development [11, 12].

From our motivating example and review of associated literature we have identified the following key requirements for a tool to assist mobile app developers that should:

- R1. Automate generation of the boilerplate code required for data-intensive mobile apps.
- R2. Design a tool for professional mobile app developers.
- R3. Generate apps that provide the full capabilities of the underlying platform i.e. provide flexibility in the generated apps.
- R4. Enable rapid prototyping of a fully functioning mobile app.
- R5. Produce prototypes that can be refined into the final app.
- R6. Provide multiple abstraction levels for modeling the different concerns of a mobile app i.e. navigation flow and UI composition.

4. Mobile App Concepts

App development involves different levels of abstractions that build upon the previous level and allow developers to reason about different aspects of a mobile app. Building tools at the appropriate level of abstraction requires making trade-offs between flexibility and productivity as the higher levels are harder to modify. Each level of abstraction is discussed below from the lowest level to the highest.

- *Low Level Platform APIs.* This is the lowest level of abstraction provided by a mobile platform. Being the lowest level, it permits the maximum flexibility as developers can modify every aspect of the platform. Typically the rest of the platform's APIs are implemented using this low level API.
- *Base Components and APIs.* Platform vendors provide components from which app developers build their apps such as buttons, classes for creating screens and fetching data from webservices or for interacting with platform features such as sensors. The majority of a mobile app is currently developed at this level of abstraction as it strikes the appropriate trade-off between flexibility and productivity.
- *High Level App Concepts.* These are the concepts that non-developers can reason about or end users can experience. Implementing these concepts needs significant development work, requiring the use of multiple

lower level APIs. Example concepts at this level are screen, call to webservice, maps etc. Current literature on DSLs for the mobile app domain focus on capturing the abstractions present at this level but lack the capabilities for the lower level abstractions prohibiting flexibility in created apps, or fail to provide higher level views of the entire app to developers. The concepts at this level can often be mapped across platforms.

- *App Concerns.* This is the highest level of abstraction and is focused on viewing and manipulating app wide concerns such as navigation flow and data modeling. Model driven engineering approaches to mobile app development often work at this level of abstraction.

Mobile app developers have to address all four levels of abstraction and work in a rapidly changing environment where high level abstractions have yet to be defined for new app features – new features do not have stable or well defined abstractions suitable for model driven approaches. For example, mobile platform vendors frequently update their design guidelines but may not update their APIs at the same time requiring developers to build custom components from scratch. Consider the example of using a UML tool (App Concerns) [27] to model a data-intensive mobile app which includes a new UI navigation pattern such as a Drawer [4]. Implementing a Drawer requires using High Level App Concept of a list, Base Components and APIs for displaying text and images, and Low Level Platform APIs to handle custom animation and styles. Developers cannot always wait for the platform to implement a feature they need or for a 3rd party library to become available. Essentially all 4 levels of abstraction have to be considered as first class citizens when building a model driven tool for the mobile app domain due a rapidly changing environment.

5. Our Approach

Our approach RAPPT is a tool that provides multiple views for specifying the concepts of a mobile app and generates working prototypes that form the scaffolding of the final app. The focus of our work has been on building a tool that targets professional app developers and assists them in the early

[4]https://www.google.com/design/spec/patterns/navigation-drawer.html

stages of development by providing them with a DSVL to specify high level app features and then add extra details using a DSTL. After the specification has been completed RAPPT generates the source code for a single platform, a working Android app, to which the developer adds the final polish. The major steps involved in using RAPPT to generate a new project are shown in Figure 2. The core aspects of our approach *Generate an App for a Single Platform*, use an *Empirically Derived Meta-model* and use *Multiple Views with Overlapping Abstraction Levels* will be described in detail below.

5.1. How RAPPT Works

This section outlines how developers use RAPPT to build a mobile app and the numbers below map to Figure 2.

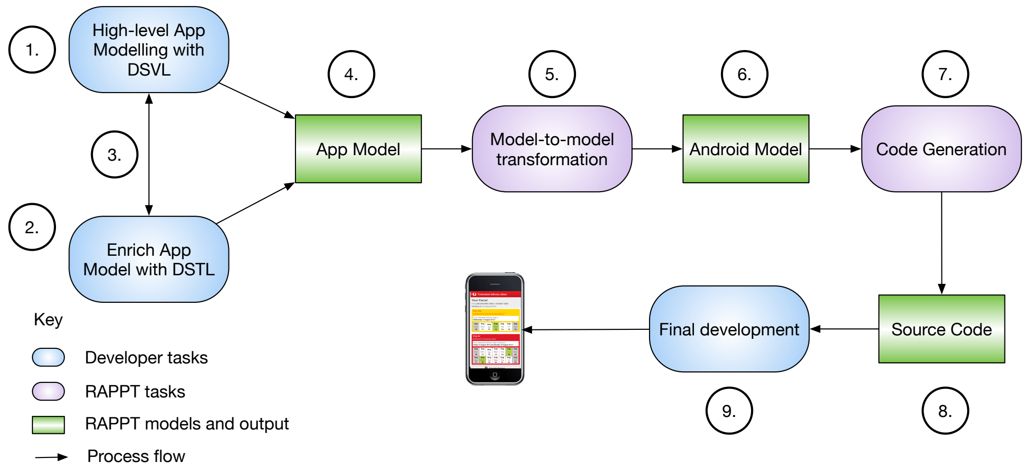


Figure 2: RAPPT assists software developers by generating the initial architecture for their app.

1. Developers start using RAPPT by describing the high level structure of an app using the DSVL. This includes specifying the number of screens in the app, the navigation flow between screens and major features per screen such as Google maps.

2. Developers can then switch to using the DSTL to provide additional details to the mobile app that are not provided by the DSVL. These features include specifying details about a web service for fetching data,

defining the data schema, specifying and configuring authentication, and specifying the information to display in specific UI elements such as rows in a list.

3. Both the DSVL and DSTL update an App Model permitting developers to switch between the interfaces as they proceed through the modelling process. Developers can also view the source code that will be generated from their model.

4. RAPPT then performs a model-to-model transformation to convert the App Model to an Android Model. This process fills in additional details about the Android app to ensure that the generated app closely resembles that of written by a developer. These features include adding the correct permissions, selecting the correct classes from the Android SDK i.e. Fragment vs Activity, externalising strings for internationalisation and adding the appropriate dependencies to the project.

5. The Android Model contains all of the information required for generating the source code. Future work can explore using this model to identify violations of mobile app development best practices. The Android Model also contained fields that were not to be used in the generated code such as the directory structure following Android conventions.

6. Next, RAPPT maps code templates to the Android Model and generates source code.

7. The generated source code from RAPPT contains the scaffolding for the mobile app that is ready to be run on the device. This code can be compiled and tested immediately after generation.

8. Developers are needed to finish off the generated app by adding business logic and styling.

5.2. Generate an App for a Single Platform

One of the key motivations of our approach is requirement *R3. Generate apps that provide the full capabilities of the underlying platform i.e. provide flexibility in the generated apps.* To achieve this we focused on generating code for a single platform which meant we did not need to handle discrepancies between platforms and could generate code that adhered to the platform's UI guidelines. We designed our code generator to produce mobile apps that could be compiled and deployed to a device without modification satisfying the requirement of *R4. Enable rapid prototyping of a fully functioning mobile app.* This enables developers to produce the first prototype quickly and can gather feedback on the navigation flow for the app during the initial client meeting. As mentioned above app development begins with the UI and UX and then moves onto the development tasks. To enable a smooth transition from the prototyping stage and to ensure there is no wasted effort we designed the generated apps in a way that it forms the scaffolding for the final app. Once the initial prototyping stage was complete developers take the generated app and build the rest of the app on top of what was generated enabling RAPPT to satisfy requirement *R5. Produce prototypes that can be refined into the final app.*

5.3. Empirically Derived Meta-model

Our approach leverages techniques from Model Driven Development (MDD) especially the field of Domain Specific Languages in the generation of mobile apps. MDD's use of models to abstract away implementation details is well suited to address the requirement *R1. Automate the boilerplate code required for data-intensive mobile apps.* Underpinning both our DSVL and DSTL is a shared meta-model, shown in Figure 3, for data-intensive mobile apps that contains the core concepts required to model and generate a working mobile app. It is the shared meta-model that enables the user to switch between editing the DSVL and the DSTL, and that provides developers with the productivity boost by providing them with high level abstractions.

There are two main reasons why we decided to derive a meta-model. First, we wanted to identify the smallest number of concepts that can be mapped to source code that addresses the technical domain concerns of mobile apps [13]. Second, our focus was on identifying concepts used frequently in real apps rather than assuming all concepts offered by a mobile app SDK are essential for rapid generation.

All of the abstractions are available in the DSTL but only some of the concepts are available in the DSVL as they are not all needed to specify the high level concerns of a mobile app. As shown in Figure 3, the concepts of web services, data fields and the data model can only be specified by the DSTL due to the requirement to accept input from the user i.e the URL for the web service requires text input. The requirement *R2. Design a tool* *aimed at professional mobile app developers*, meant that the meta-model also had to include concepts that developers use to describe mobile apps.

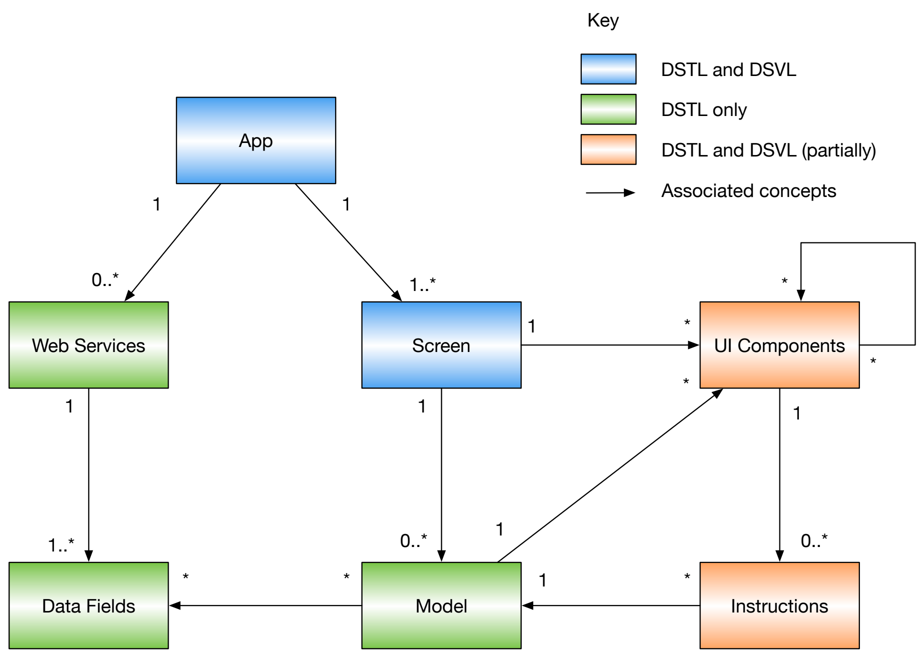


Figure 3: RAPPT's meta-model underpinning the DSVL and DSTL. Coloured boxes show which concepts are available in the DSVL and DSTL.

Each app has specific settings unique to that app such as the first screen is to be shown when the app launches. These concerns are captured in our meta-model by the concept of *App*. In order for an *App* to be able to model data-intensive mobile apps, it should include two major concepts *Web Services* which provide the contents for the app, and *Screens* that display the data to the end user. *Data Fields* represent the data that is returned from a *Web Service* and forms the data model of the mobile app. The data model for an app *Model* captures the structure of the data and what is shown to the user. *Screens* are made up of *UI Components* that display parts of the data model and both may trigger an *Event*. *Events* may be fired from a button or be triggered with the screen loads and contain *Instructions* which perform a task; most commonly a call to a web service to fetch data to display or render to the screen.

To create the meta-model we analyzed 30 data-intensive mobile apps and from these extracted the core concepts. A data-intensive app was considered an app that 1) primarily relied on data to provide its functionality, 2) was a complete app rather than a live widget running on the home screen and 3) was not categorized as Games. Studying these apps, we built a primitive version of the meta-model which we used to generate new apps. When we identified concepts that the tool could not generate we added the concepts to the meta-model and then extended RAPPT. In this manner, we were able to follow an iterative process to build the meta-model. These concepts formed the basis for the concepts in our DSTL that we have described previously [2] and informed the design of our DSVL. List of the studied apps is available online[5].

5.4. Multiple Views with Overlapping Abstraction Levels

Software engineers utilize higher level abstractions to hide the unnecessary details and hence focus on the problems at hand. As such we needed to ensure that RAPPT could satisfy requirement *R6. Provide multiple abstraction levels for modeling the different concerns of a mobile app i.e. navigation flow and UI composition.*. For each of the views discussed in Section 4, different abstractions are needed. These abstractions could conflict with each other. Navigation and UI layout views both need a screen but require different information. The navigation view is concerned with how a screen is connected to other screens whereas the UI layout view provides precise configuration of UI components.

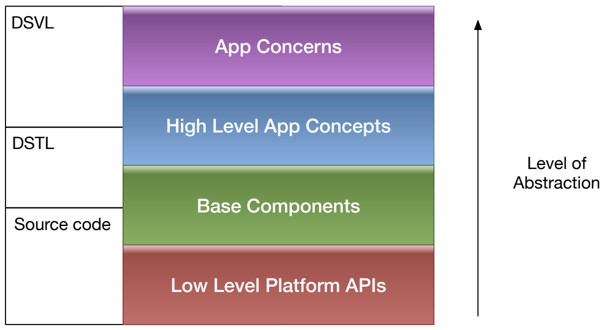


Figure 4: Levels of abstraction in a mobile app.

[5]https://github.com/ScottyB/analysed-apps

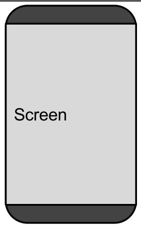
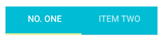
Our approach provides three views for developing mobile apps consisting of overlapping levels of abstraction to address these conflicting scenarios: A DSVL for high level app functionality, a DSTL for providing additional details not available in the DSVL and access to the target platform for creating custom app functionality. Figure 4 provides an over view of these abstraction levels, their composing elements, and their relations.

Underpinning both the DSTL and the DSVL is a meta-model of an Android app that is used to generate code once the initial modeling stage is complete. The DSVL includes concepts for modeling the high level concepts (*App Concerns*) of data-intensive apps such as navigation flow and the Data Model. Included in the DSVL are a number abstractions that are categorized as *High Level App Concepts*, and can be enhanced using the DSTL. For example, the concept of a screen is a *High Level App Concepts* that is present in the DSVL so that the navigation flow can be modeled but the DSTL is required to specify what will be displayed on the screen or which webservices it calls. A summary of the concepts present in the Visual Language is shown in Table 1. The rational for choosing visual notations was to choose the representations that closely relate to the concepts in the meta model and the target platform (here Android operating system). Only a few concepts have been added to the Visual Language as the focus of this paper is on the multi-view approach to mobile app development, rather than the complete visual app builder.

The DSTL also includes abstractions from the *Base Components* category. These abstractions are added to enhance the specification for screens and to be able to model webservices for fetching data to render to the screen. Examples in the DSTL are input fields, images and keywords for specifying webservices. For highly custom features and concepts that cannot be modeled by any of the previous stages developers have to use the *Low Level Platform APIs* in with the generated code. Once the developer has finished modeling RAPPT generates code for the developer to modify which provides maximum flexibility to the developer as they are only limited by what the target platform provides.

[6] https://developers.google.com/maps/documentation/android/

Table 1: Example visual elements that make up RAPPT's Visual Language

Concept	**Notation**	**Description**
Screen		Represents a screen displayed on a mobile device as seen by the end user.
Button Navigation	- - - - ➔	Represents navigation from one screen to another by clicking on the UI component Button.
Map		Displays a Google Map[6]
Tabbar		Represents the Mobile navigation UI pattern, Tabbar.

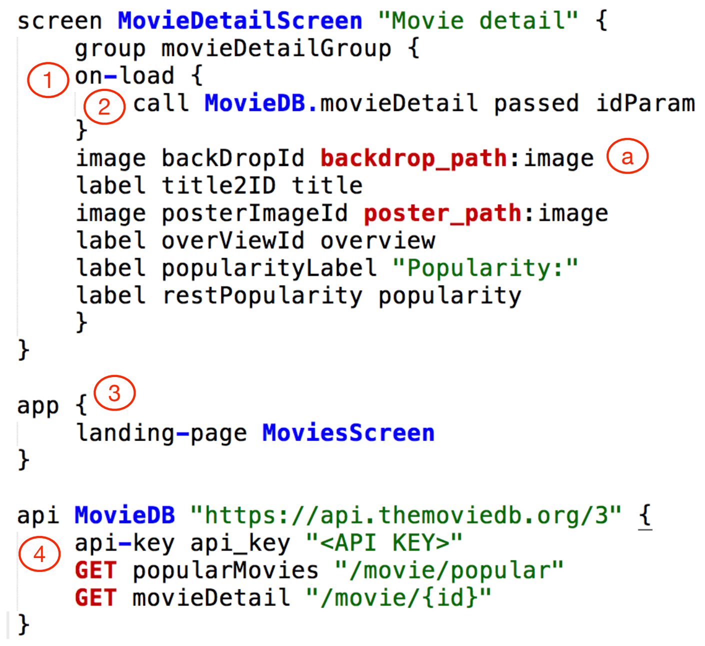
5.5. DSTL Grammar and Syntax

The grammar for the DSTL is derived from the meta-model and the full definition is available online[7]. Both the app and screen concepts from the meta-model have a direct mapping to concepts in the DSTL. All of the other concepts in the meta-model represent an abstract instance of the entities in the DSTL. For example, Web Services in the meta-model includes the following abstractions *api* for specifying a datasource, *api-key* for authentication, and *GET,POST* for making calls to a webservice. Example code from the DSTL used in RAPPT can be seen in Figure 5.

6. Implementation

Developers interact with RAPPT through a web interface implemented using standard web technologies (i.e. HTML 5, CSS and Javascript). The

[7]https://github.com/ScottyB/rappt/blob/master/grammar.g4



```
screen MovieDetailScreen "Movie detail" {
    group movieDetailGroup {
    on-load {
        call MovieDB.movieDetail passed idParam
    }
    image backDropId backdrop_path:image
    label title2ID title
    image posterImageId poster_path:image
    label overViewId overview
    label popularityLabel "Popularity:"
    label restPopularity popularity
    }
}

app {
    landing-page MoviesScreen
}

api MovieDB "https://api.themoviedb.org/3" {
    api-key api_key "<API KEY>"
    GET popularMovies "/movie/popular"
    GET movieDetail "/movie/{id}"
}
```

Figure 5: Sample DSTL code for loading data from an API and showing the results to the screen: 1) an event handler for catching the on screen load event, 2) a call to an API with parameters, 3) the landing page for the generated app and 4) definition of the API.p

App Model was implemented as a JSON object and was shared between the server and the client. Using JSON as the model format meant that it could easily be sent to and from the server as many web apps use JSON as a data transmission format. The DSTL Processor was implemented using the ANTLR compiler compiler which compiled the DSTL into an internal representation of the App Model. Errors that were generated on the server were sent to the client and displayed to the user. We implemented the server side code in Java and the code templates using String Template. Additional details about the tool implementation can be found online [8].

We made the decision to build RAPPT as a web based tool to simplify the process of getting started i.e. there is nothing to download and setup to use the tool. This also meant that the tool was available anywhere for rapid prototyping. The DSTL Processor had to be developed on the serverside due to the available libraries for a compiler compiler.

[8]https://github.com/ScottyB/rappt/blob/master/rappt-tool-guide.pdf

7. User Evaluation

We have conducted a user evaluation of RAPPT for mobile application development. Our primary aims in this user evaluation was to evaluate user acceptance and examine how using RAPPT can speed up application development for experienced software developers both with and without mobile application development experience. All of the resources for the evaluation task, survey questions and results are available online [9]. The details of this user study follows.

7.1. Experiment setup and tasks

Participants were first asked to complete a demographic survey before starting the experiment. They then watched instructional videos demonstrating how to use the online editor and visual model. These learning videos also showed how to construct a small Android app using both the visual editor and AML. By using learning videos we were able to reduce bias and run multiple sessions with different participants. On completion of the learning videos participants were asked to fill out a survey to ascertain their confidence level with building Android apps with RAPPT. The next stage involved an evaluation task where participants were asked to build three screens for an app that displayed data from the MovieDB API similar to the motivation example of section 2. Participants used their own computers and accessed RAPPT online.

Participants were encouraged to use the provided samples. Once participants felt that they had completed the task, their program was downloaded and run on instructors' machine. Apps created by the participants were then installed on a device and run to ensure that there were no runtime issues caused by the generated code. It would also allow participants to see the application they have developed being instantly installed and used on a mobile device.

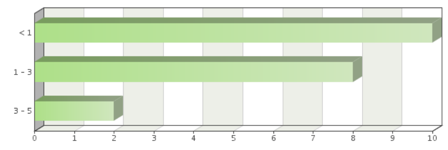
Upon finishing the experiment, a matching questionnaire was handed to each participant. This questionnaire was composed of 16 questions with 5 point Likert scale ranging from *Strongly Disagree* to *Strongly Agree*, and 9 open-ended questions capturing their experience of using RAPPT to build Android apps. We followed the positive questionnaire design approach as

[9]https://github.com/ScottyB/rappt-eval/tree/master/user-eval

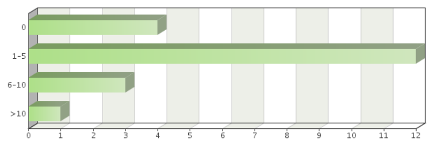
suggested by Sauro et al. [28]. This would help us analyze the results faster, avoid accidental mistakes and have a more consistent set of questions.

7.2. Participants

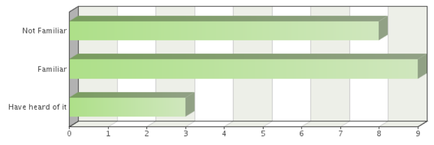
The participants of this user evaluation were selected from software developers and researchers at Swinburne University of Technology (Australia). Overall 20 participants were recruited (17 male, 3 female). Our demographics questionnaire included six questions to capture participants' background and their experience in mobile application development particularly experience developing apps for Android operating system. These demographics questions and participant answers are provided by Figure 6.



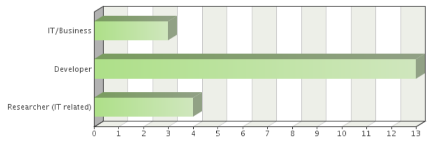
(a) D1 - How many years experience do you have in mobile app?



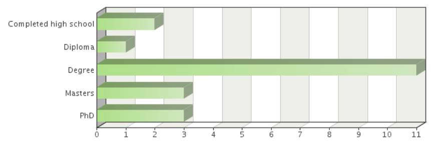
(b) D2 - How many apps have you built?



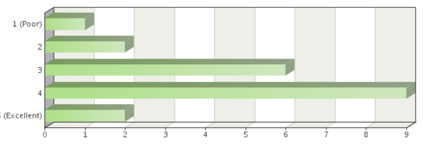
(c) D3 - How familiar are you with the Android Annotations library?



(d) D4 - What is your background?



(e) D5 - What is the highest qualification you have?



(f) D6 - How would you rate your skills to be able to build a simple data driven mobile app?

Figure 6: User demographic questions and participants' responses.

7.3. Results and discussion

Out of 20 participants, only one was *not* able to finish the experiment successfully. This was due to the participant's lack of interest in mobile app development and has been reflected in the questionnaire responses. It took the participants approximately 90 minutes on average to finish the tasks and answer the questions. Table 2 presents a selection of questions from the questionnaire. It also depicts the frequency of participant responses to each question.

The overall response was positive with 80% of participants agreeing or strongly agreeing to 7 out of the 9 questions. These results confirm that our approach is suitable for use by professional developers

Q2 received the strongest result with 65% of participants Strongly Agreeing to the icons in the DSVL being easy to understand. This confirms our current choice of icons for the concepts present in a data-intensive mobile app. We are constantly updating the DSVL based on our feedbacks. This includes updating the icons to match target platforms, and including more meta model and fine grained concepts in the DSVL.

An interesting finding from the survey was the results to question **Q4** on RAPPT enabling users avoid making mistakes. It demonstrates a close to normal-distribution spread. The answers and comments on the questionnaire indicated needs for improving error handling mechanisms as most participants had experience with major software development IDEs. Some participants mentioned that they would have preferred a more real-time error handling mechanisms. A possible way to improve the error handling is to move more concepts from the DSTL to the DSVL – visual languages are not as susceptible to developer error as textual languages are. Improving the overall robustness of the tool is another way to address the issue of poor error reporting.

Answers to **Q5** were also non-committal with 40% of participants being neutral. The concepts in RAPPT cover the main concepts required for modeling data-intensive apps but are not exhaustive. For example the concept of a navigation pattern is present although not every navigation UI pattern is supported. This is one reason why the results may not be strongly one way or another. Clarifying the question by indicating data-intensive apps rather than all mobile apps would have removed some confusion.

Participant responses to question **Q6** indicate the overall acceptance of the approach. 95% of the participants had positive views on the usefulness of the approach (60% Strongly Agree and 35% Agree). Same is true for responses to question **Q8**.

7.4. Threats to Validity

Although we have tried our best to reduce threats to validity for the experiment, there are certain threats with regards to participant affiliations

Table 2: Sample questions of the questionnaire. Likert points have been given score of 1 to 5 representing *Strongly Disagree* to *Strongly Agree*.

No.	**Question**	**Frequency (%)**				
		1	**2**	**3**	**4**	**5**
Q.1	It was easy to use RAPPT.	0	5	15	50	30
Q.2	It is easy to understand what each icon represents.	0	0	10	25	65
Q.3	It was easy to load data from an API and render it to the screen.	0	0	15	55	30
Q.4	It is easy to avoid making errors or mistakes.	5	20	35	30	10
Q.5	The concepts in RAPPT are - sufficient for modelling a mobile app.	0	5	40	35	20
Q.6	RAPPT is useful for mobile app development.	0	0	5	35	60
Q.7	Using RAPPT is more efficient than starting with a raw Android project.	5	5	10	35	45
Q.8	You are satisfied with using RAPPT.	0	0	10	40	50

and background. In this section we outline some of these threats.

Internal Validity Our participants have been recruited from software engineers and developers at Swinburne Software Innovation Laboratory. Their affiliation may have introduced bias in their responses. Also some participants knew researchers which may have had effect on their responses to the questionnaire. During the design of RAPPT, we have considered the needs of expert as well as novice mobile app developers. The majority of our participants (18 out of 20) had less than three years experience developing mobile apps, and 10 participants had less than one year developing mobile apps. This may result in some bias towards novice app developers. However, developers or students first getting started with mobile app development can benefit from a tool like RAPPT.

External Validity The User Evaluation involved 20 participants which is sufficient for our purposes but not for statistical analysis. Evaluating RAPPT with more developers on a wide range of applications would further improve confidence that RAPPT is generally applicable for developers. Another threat to validity is best summed up with a comment from one of the participants ... I need more time to play around with RAPPT... Participants built one app taking approximately 1 hour to complete and are likely to give different responses after using RAPPT extensively. It would be interesting to see how the answers from the participants change after using RAPPT for an extended period of time on multiple projects. The participants were asked to complete one task. An improvement to the evaluation task would involve selecting a range of apps with different functionality and purposes. Evaluating RAPPT on additional apps would also help expose gaps in the model and the language which could be used to inform the implementation.

Construct Validity In our evaluation we asked the participants to build an app which the authors verified against a set criteria available online [10]. The participants were not asked to import the generated projects into an IDE which would be necessary in a commercial environment. The simple task that developers were asked to complete did not require the participants to modify the generated code. Thus, not every step required for using RAPPT was evaluated. Future work would require participants to download and extend the generated output by adding an additional feature to the application.

8. Conclusion

We have introduced RAPPT, an approach and tool support for rapid design and prototyping of mobile applications. RAPPT provides a DSVL and a DSTL for mobile application development. It also utilizes multiple views and abstractions levels of mobile applications to help developers be more efficient in prototyping various apps and at the same time, have maximized customization ability. In addition, we have presented 4 levels of abstraction present when building mobile apps and cater for the different levels in our approach. We have evaluated RAPPT using a user study involving 20 developers and researchers. The results of this evaluation demonstrated acceptance of the approach among software and mobile app developers. From the responses to our user evaluation RAPPT can be used as a starting point

[10]https://github.com/ScottyB/rappt/blob/master/criteria.md

to get to the first version of the code base up and running rapidly. Although we have not evaluated the time it took our users to develop the sample application as opposed to manually implementing the features in the code, we have received qualitative responses that confirm using RAPPT improves productivity. Future work will look at quantifying how much of a productivity boost RAPPT provides in comparison with implementing the feature from scratch and using other model driven approaches. Another avenue of future work would be to build up the DSVL to include more concepts and to provide a constraint checking mechanism. Validating and verifying the visualisations prior to code generation will also be an interesting area of future work. Finally, addressing the issue of round trip engineering is another area that would improve the current work.

References

[1] F. Bentley, E. Barrett, Building Mobile Experiences, The MIT Press, 2012.

[2] S. Barnett, R. Vasa, J. Grundy, Bootstrapping mobile app development, in: Proceedings of the 2015 IEEE/ACM International Conference on Software Engineering (ICSE 2015), IEEE, 2015, pp. 305–306.

[3] F. T. Balagtas-Fernandez, H. Hussmann, Model-driven development of mobile applications, in: Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on, IEEE, 2008, pp. 509–512.

[4] J. Danado, F. Paternò, A prototype for eud in touch-based mobile devices, in: Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on, IEEE, 2012, pp. 83–86.

[5] B. Athreya, F. Bahmani, A. Diede, C. Scaffidi, End-user programmers on the loose: A study of programming on the phone for the phone, in: Visual Languages and Human-Centric Computing (VL/HCC), 2012 IEEE Symposium on, IEEE, 2012, pp. 75–82.

[6] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, et al., Scratch: programming for all, Communications of the ACM 52 (11) (2009) 60–67.

[7] D. Wolber, App inventor and real-world motivation, in: Proceedings of the 42nd ACM technical symposium on Computer science education, ACM, 2011, pp. 601–606.

[8] R. Francese, M. Risi, G. Tortora, Iconic languages: Towards end-user programming of mobile applications, Journal of Visual Languages & Computing 38 (2017) 1–8.

[9] C. Rieger, Business apps with maml: a model-driven approach to process-oriented mobile app development, in: Proceedings of the Symposium on Applied Computing, ACM, 2017, pp. 1599–1606.

[10] M. Naab, S. Braun, T. Lenhart, S. Hess, A. Eitel, D. Magin, R. Carbon, F. Kiefer, Why data needs more attention in architecture design - experiences from prototyping a large-scale mobile app ecosystem, in: Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, 2015, pp. 75–84. doi:10.1109/WICSA.2015.13.

[11] A. Shye, B. Scholbrock, G. Memik, Into the wild: studying real user activity patterns to guide power optimizations for mobile architectures, in: Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture, ACM, 2009, pp. 168–178.

[12] S. Barnett, R. Vasa, A. Tang, A conceptual model for architecting mobile applications, in: Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on, 2015, pp. 105–114. doi:10.1109/WICSA.2015.28.

[13] S. Barnett, Extracting technical domain knowledge to improve software architecture.

[14] D. Steiner, C. Ţurlea, C. Culea, S. Selinger, Model-driven development of cloud-connected mobile applications using dsls with xtext, in: Computer Aided Systems Theory-EUROCAST 2013, Springer, 2013, pp. 409–416.

[15] H. Heitkötter, T. A. Majchrzak, H. Kuchen, Cross-platform model-driven development of mobile applications with md 2, in: Proceedings of the 28th Annual ACM Symposium on Applied Computing, ACM, 2013, pp. 526–533.

[16] O. Le Goaer, S. Waltham, Yet another dsl for cross-platforms mobile development, in: Proceedings of the First Workshop on the Globalization of Domain Specific Languages, GlobalDSL '13, ACM, New York, NY, USA, 2013, pp. 28–33. doi:10.1145/2489812.2489819.
URL `http://doi.acm.org/10.1145/2489812.2489819`

[17] H. Behrens, Mdsd for the iphone: developing a domain-specific language and ide tooling to produce real world applications for mobile devices, in: Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion, ACM, 2010, pp. 123–128.

[18] I. Madari, L. Lengyel, T. Levendovszky, Modeling the user interface of mobile devices with dsls, in: Proc. of the Computational Intelligence and Informatics 8th International Symposium of Hungarian Researchers, 2007, pp. 583–589.

[19] X.-S. Li, X.-P. Tao, W. Song, K. Dong, Aocml: A domain-specific language for model-driven development of activity-oriented context-aware applications, Journal of Computer Science and Technology 33 (5) (2018) 900–917.

[20] G. Taentzer, S. Vaupel, Model-driven development of mobile applications: Towards context-aware apps of high quality., in: PNSE@ Petri Nets, 2016, pp. 17–29.

[21] D. Bolchini, A. Faiola, The fusing of paper-in-screen: Reducing mobile prototyping artificiality to increase emotional experience, in: Design, User Experience, and Usability. Theory, Methods, Tools and Practice, Springer, 2011, pp. 548–556.

[22] M. De Sá, L. Carriço, A mobile tool for in-situ prototyping, in: Proceedings of the 11th International Conference on Human-Computer Interaction with Mobile Devices and Services, ACM, 2009, p. 20.

[23] A. P. Jørgensen, M. Collard, C. Koch, Prototyping iphone apps: realistic experiences on the device, in: Proceedings of the 6th Nordic Conference on Human-Computer Interaction: Extending Boundaries, ACM, 2010, pp. 687–690.

[24] M. Challenger, S. Demirkol, S. Getir, M. Mernik, G. Kardas, T. Kosar, On the use of a domain-specific modeling language in the development of multiagent systems, Engineering Applications of Artificial Intelligence 28 (2014) 111–141.

[25] A. A. Shah, A. A. Kerzhner, D. Schaefer, C. J. Paredis, Multi-view modeling to support embedded systems engineering in sysml, in: Graph transformations and model-driven engineering, Springer, 2010, pp. 580–601.

[26] H. Zhao, L. Apvrille, F. Mallet, Multi-view design for cyber-physical systems, in: PhD Symposium at 13th International Conference on ICT in Education, Research, and Industrial Applications, 2017, pp. 22–28.

[27] F. A. Kraemer, Engineering android applications based on uml activities, in: Model Driven Engineering Languages and Systems, Springer, 2011, pp. 183–197.

[28] J. Sauro, J. R. Lewis, When designing usability questionnaires, does it hurt to be positive?, in: Proceedings of the SIGCHI Conference on Human Factors in Computing Systems, CHI '11, ACM, 2011, pp. 2215–2224.